THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 8, October 1982
Printed in U.S.A.

Implementing and Testing New Versions of a Good, 48-Bit, Pseudo-Random Number Generator

By C. S. ROBERTS

(Manuscript received October 27, 1981)

In this paper we describe the design, testing, and use of drand48—a good, pseudo-random number generator based upon the linear congruential algorithm and 48-bit integer arithmetic. The drand48 subroutine is callable from C-language programs and is available in the subroutine library of the UNIX operating system. Versions coded in assembly language now exist for both the PDP-11 and VAX-11 computers; a version coded in a "portable" dialect of C language has been produced by Rosler for the Western Electric 3B20 and other machines. Given the same initialization value, all these versions produce the identical sequence of pseudo-random numbers. Versions of drand48 in the assembly language of other computers or for other programming languages clearly could be implemented, and some output results have been tabulated to aid in testing and debugging such newly coded subroutines. Timing results for drand48 on the PDP-11/45, the PDP-11/70, the VAX-11/750, and the VAX-11/780 are also presented and compared.*

I. INTRODUCTION

The work described in this paper arose when one day the author found himself in need of a good, pseudo-random number generator that would execute and produce identical results on two different computers (in this case, the 16-bit PDP-11 and the 32-bit VAX-11, both manufactured by Digital Equipment Corporation). Good, pseudo-random number generators often require multiple-precision arithmetic; hence, to achieve speed they are usually implemented in assembly language and are dependent upon the word length of the computer. If

* *UNIX* is a trademark of Bell Laboratories.

this is not enough of a barrier to portability, add the fact that the recoding of a pseudo-random number generator for a different computer is an error-prone endeavor. The author has himself found bugs in several pseudo-random number subroutines that were supposedly "correctly" coded. (The author admits to being facetious, but do such bugs make the output from a generator more or less random?)

This paper does not present a magical method to eliminate these difficulties. However, it does present enough data and intermediate results so that a person may code on any computer a good generator based upon 48-bit integer arithmetic and then begin to test the new version for bugs. In addition, we describe proper usage of routines drand48, lrand48, mrand48, erand48, nrand48, and jrand48—C-language callable functions to generate pseudo-random numbers—currently available at Bell Laboratories in the subroutine library of the *UNIX** operating system for the Western Electric 3B20, the PDP-11, tbe VAX-11, and other computers.

The pseudo-random number generator considered in this paper is based upon the well known linear congruential algorithm, e.g., see Section 3.2.1 of Knuth.[1] The next number in the pseudo-random sequence is generated according to the formula

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\, m} \quad n \geq 0.$$

We choose the value of m to be 2^{48}; hence, 48-bit integer arithmetic is required. The values of the multiplier a and the addend c are chosen as follows:

$$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$$

$$c = \mathrm{B}_{16} = 13_8.$$

While other equally good choices for m are clearly possible, we chose 2^{48} for the following reasons, based upon matters of taste and judgment. The word length of many popular computers is a multiple of 16 bits; bence, it seemed wise and convenient to choose $\log_2 m$ to be an integer multiple of 16. A period of 2^{48} for the generated pseudo-random sequence seemed long enough for most purposes, while 2^{32} seemed dangerously short. Assuming it takes 10^{-4} second to generate a pseudo-random number (see the timing results in Section IV), a period of 2^{48} corresponds to 893 years, while 2^{32} would only require 119 hours for tbe complete cycle to be generated. The argument for 48 bits becomes even more compelling if we assume that a future processor or customized bardware might be a factor of 10 or 100 faster.

The value of a, chosen above, was one of the multiplier values judged by Coveyou and MacPherson[2] to be satisfactory according to the "spectral test," one of the most demanding of the canonical tests

for overall pseudo-random number-generator quality. The above values of a, c, and m ensure that the generated pseudo-random sequence will have the maximum possible period, i.e., 2^{48} (see Theorem A, Section 3.2.1.2 of Knuth[1]). The testing done by Coveyou and MacPherson[2] was demanding enough and the properties of pseudo-random sequences based upon the linear congruential algorithm are well enough understood[1] that further statistical testing of the generator was deemed unnecessary.

The linear congruential algorithm with these choices for the parameters yields a generator that exceeds the requirements of most users and is much better than many generators in common use today. Obviously, this is not the "optimum" pseudo-random number generator; it will not meet all the requirements of any potential user, and the author makes no such claims. However, in more than four years of use in numerous programs at Bell Laboratories, it has served well, even in situations where other generators have failed.

II. PSEUDO-RANDOM GENERATOR RESULTS

The pseudo-random sequence generator described and specified in Section I was coded in assembly language three times—once on the PDP-11 using the 16-bit integer arithmetic instructions "mul" and "add,"[3] once on the PDP-11 using the 64-bit floating-point arithmetic instructions "mulf" and "addf,"[3] and once on the VAX-11 using the 32-bit integer extended multiply and add instruction "emul."[4] After a bit of debugging, all three independent versions finally produced the identical sequence of 48-bit pseudo-random integers. For purposes of comparison with future newly coded subroutines, a portion of that sequence is presented below (in hexadecimal) as Table I.

Since it is common practice to treat the output from a pseudo-random number generator as a pure binary fraction, thus yielding a uniform distribution over the interval 0 to 1, the numbers in Table I were so treated and then multiplied by 4096 to yield the decimal

Table I—A portion of the pseudo-random sequence X_i of 48-bit integers

1234ABCD330E	657EB7255101	D72A0C966378	5A743C062A23	72534ABF62F2
5195D97A8D15	E2ECF94AEFFC	03FD3CD49657	9586EFCA2D16	28CC61DEF669
623B341D40C0	B0E5A9A111CB	0F11604F57A	E65CDA1020FD	29DE25BD59C4
28B8E8F5507F	8876EDD9601E	9AA93190E0D1	952BC3577F08	451CD3C24673
63F661075102	4a1C4CBD49E5	aE0C7218348C	4C6C2C9427A7	135676A8EC26
67ACF11EB039	Da7D1EF03E50	F124D606681a	A9AF4526958A	D8a2A2FFA7CD
00B48E98A054	765E7C77BBCF	8858368AF12E	C9B2484004A1	43FF29D69E98
FB95A6FE16C3	4E897866E312	99D1A468DAB5	9BD4C9FFBD1C	3662639AACF7

Table II—Pseudo-random integers Y_i corresponding to the X_i in Table I

291	1623	3442	1447	1829	1305	3630	63	2392	652
1571	2830	241	3685	669	651	2183	2474	2386	1105
1599	1201	3040	1222	309	1658	3511	3858	2714	3467
11	1893	2181	3227	1087	4025	1256	2461	2493	870
3628	1247	622	1383	1587	2636	3086	2472	2177	1881
2672	1340	3876	1507	3866	30	2115	1117	99	2424
839	3595	243	1068	1240	3651	2040	2908	1173	3542
2767	1877	3930	3173	1542	936	1452	1230	2743	2944

integers in Table II. More precisely, each integer Y_i in Table II was computed from the corresponding value of X_i in Table I by the formula

$$Y_i = \lfloor (X_i)(2^{-48})(4096) \rfloor = \lfloor 2^{-36} X_i \rfloor.$$

For some purposes, Table II may be more convenient or appropriate than Table I.

Tables I and II should be valuable to a person attempting to code and test a new version of the generator subroutine. After initializing with $X_0 = 1234ABCD330E_{16}$, the new version, if correct, should reproduce Table I (or Table II). The fact that three independent codings of the 48-bit linear congruential algorithm gave the same results gives the author substantial confidence that Tables I and II are indeed accurate. In addition, Lawrence Rosler of Bell Laboratories has independently verified Tables I and II using both C- and assembly-language code be produced for the Honeywell H6000 series of 36-bit computers.[5]

The author has found that an empirical but effective heuristic for quickly evaluating the general quality of any pseudo-random number generator is to display the output bits on a bilevel display device, and this has been done in Fig. 1 for tbe generator defined in Section I of this paper. (A bilevel display device consists of an $n \times m$ rectangular array of dots, each of which can be made either white or black.) Figure 1 was generated using only the leftmost 32 bits of each 48-bit X_i; 2^{13} values of X_i were generated, thus giving a total of 2^{18} bits. These bits were displayed on a 512 × 512 CRT display device with a 1 being displayed as white and a 0 being displayed as black. Good quality pseudo-random number generators produce displays similar in appearance to Fig. 1—random salt and pepper effect with no visible patterning. For comparison, Fig. 2 shows the display produced by an inferior generator.* Note the prominent nonrandom patterns visible in

* The author apologizes for the generally poor quality of Figs. 1 and 2 and assures the reader that the result is much more striking and apparent when the original CRT or photographic plates are viewed. Glossy prints such as these simply do not reproduce well.

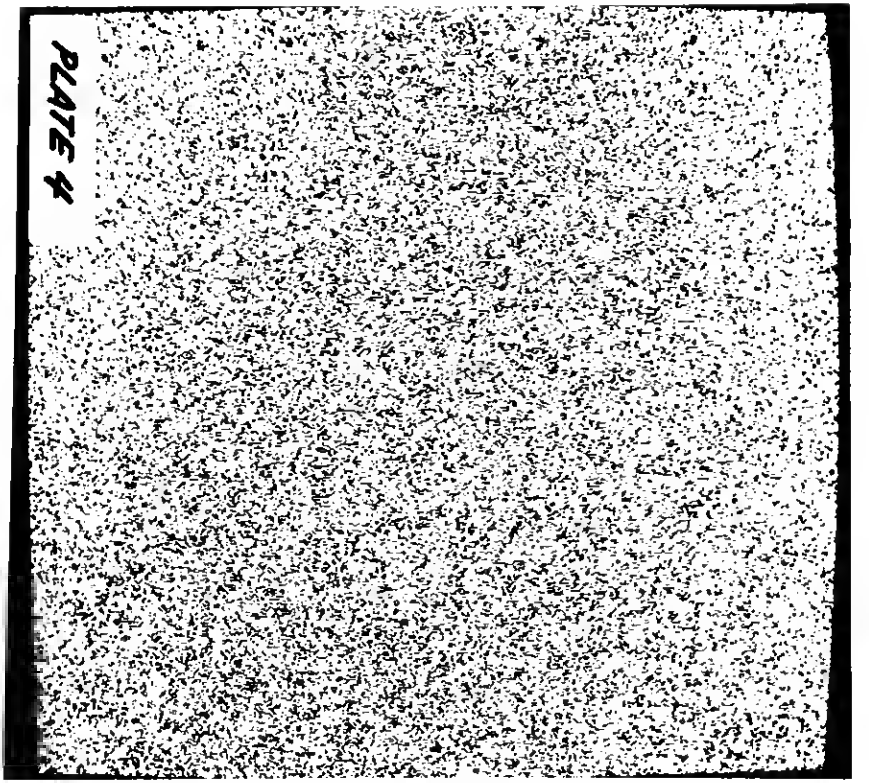

Fig. 1—Bilevel dot display generated by drand48.

Fig. 2, which was produced by a 1977 version of the library routine "rand," described in Section III of the *UNIX Programmer's Manual.* (This 1977 version used the linear congruential algorithm with $m = 2^{15}$; the version of "rand" currently in the UNIX library (1982) is better since it uses $m = 2^{31}$.)

III. SUBROUTINES FOR USE WITH THE C PROGRAMMING LANGUAGE

As an example of how the calling sequences to the pseudo-random number generator might be designed, we now describe some routines that were coded for use with the C programming language.[6] So far the author has himself implemented versions of these routines only for the PDP-11 and VAX-11 computers. However, a version for other computers that support the C language has been implemented by Rosler[7] and is now in use on the Western Electric 3B20, the Honeywell H6000, the IBM-370, and the Motorola MC68000 computers.

The pseudo-random number generator was implemented in assembly language as one subroutine having nine entry points. Six of the

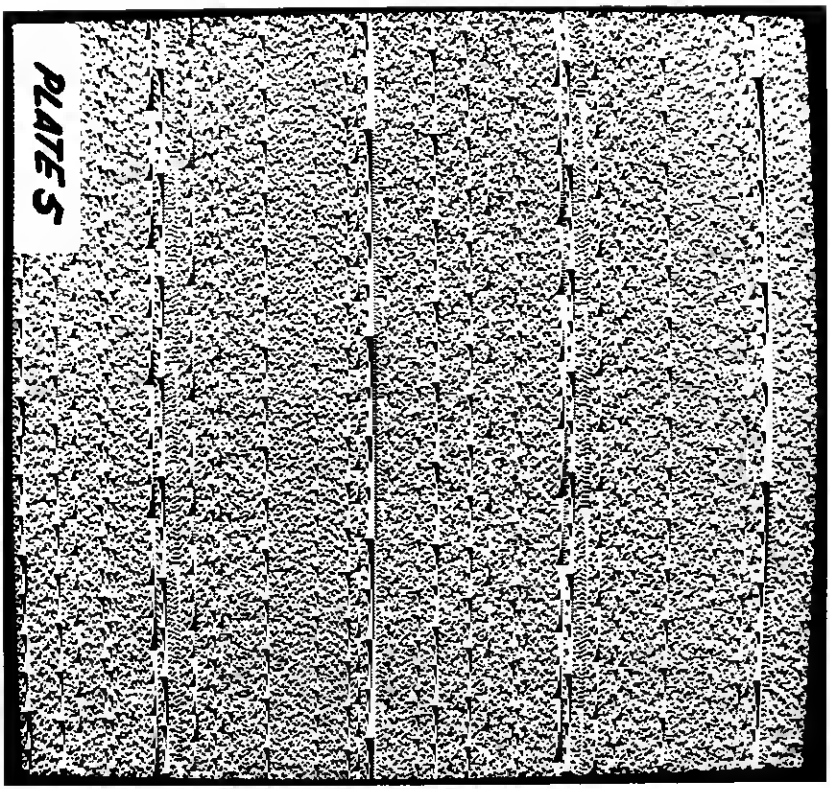

Fig. 2—Bilevel dot display generated by an inferior pseudo-random number generator.

entry points generate the next pseudo-random X_i in the sequence and then convert the leftmost bits of it into the particular type of data item desired—floating-point fraction, positive integer, or signed integer. The entry points erand48, nrand48, and jrand48 can be called without first having to invoke a special initialization entry point. Calls to the entry points drand48, lrand48, and mrand48 should be preceded by at least one call to one of the initialization entry points—either srand48, seed48, or lcong48.

3.1 Function drand48

Example C usage:

```
double fnext, drand48( );
fnext = drand48( );
```

The next pseudo-random number is returned as a non-negative binary fraction in double precision floating-point format; i.e., the value returned is $2^{-48}X_i$. The values returned are uniformly distributed over the interval [0, 1). Thus, in the above C-code example, the value range

for fnext is $0 \leq$ fnext < 1. Either srand48, seed48, or lcong48 should be invoked before calling drand48.

3.2 Function lrand48

Example C usage:

```
long int lnext, lrand48( );
lnext = lrand48( );
```

The next pseudo-random number is returned as a non-negative integer in long integer format. The long integer is formed by taking the leftmost 31 bits of X_i, i.e., the value returned is $\lfloor 2^{-17} X_i \rfloor$. In the above C-code example, the value range for lnext is $0 \leq$ lnext $< 2^{31}$. Either srand48, seed48, or lcong48 should be invoked before calling lrand48.

3.3 Function mrand48

Example C usage:

```
long int mnext, mrand48( );
mnext = mrand48( );
```

The next pseudo-random number is returned as a signed integer in long integer format. The long integer is formed by taking the leftmost 32 bits of X_i to be a signed integer in two's-complement format. Hence, the leftmost bit of X_i determines the sign of the output value. In the above C-code example, the value range for mnext is $-2^{31} \leq$ mnext $< 2^{31}$. Either srand48, seed48, or lcong48 should be invoked before calling mrand48.

3.4 Functions erand48, nrand48, and jrand48

These functions are identical to drand48, lrand48, and mrand48, respectively, in the characteristics of the data value returned. The difference is that erand48, nrand48, and jrand48 allow, and require, the calling program to provide the storage for the current 48-bit X_i value, while drand48, lrand48, and mrand48 provide this storage internally for themselves. For those programs that require only a single stream of pseudo-random numbers, drand48, lrand48, and mrand48 are a little more convenient and simpler to use. However, erand48, nrand48, and jrand48 allow multiple "independent" streams of pseudo-random numbers to be generated, i.e., subsequent numbers in each stream will not depend upon how many times the routines are called by other parts of a program to generate numbers for other streams. This property can be a big asset for certain statistical computations and for program debugging.

3.4.1 Function erand48

Example C usage:

```
double fnext, erand48( );
```

```
short int xsubi[3];
fnext = erand48(xsubi);
```

3.4.2 Function nrand48

Example C usage:

```
long int lnext, nrand48( );
short int xsubi[3];
lnext = nrand48(xsubi);
```

3.4.3 Function jrand48

Example C usage:

```
long int mnext, jrand48( );
short int xsubi[3];
mnext = jrand48(xsubi);
```

3.5 Function srand48

Example C usage:

```
long int seedval;
seedval = 0x1234ABCD;
srand48(seedval);
```

This is an initialization entry point that sets the value of X_0; the multiplier a and the addend c are set to the values specified in Section I. The leftmost 32 bits of X_0 are taken from the argument passed to srand48 when it is called (seedval in the above C-code example). The rightmost 16 bits of X_0 are arbitrarily set to $330E_{16}$. Hence, the above C-code example sets the value of X_0 to $1234ABCD330E_{16}$.

3.6 Function seed48

Example C usage:

```
short int seed16v[3], *shp, *seed48( );
seed16v[0] = 0x330E;
seed16v[1] = 0xABCD;
seed16v[2] = 0x1234;
shp = seed48(seed16v); /* pointer to previous Xi stored in shp */
        /* or alternatively, */
seed48(seed16v); /* pointer to previous Xi just ignored */
```

This is an initialization entry point that sets the value of X_0 to the 48 bits specified by the argument passed to seed48; the multiplier a and the addend c are set to the values specified in Section I. In addition, the previous value of X_i is automatically stored in a 48-bit internal buffer, used only by seed48, and the value returned is a pointer to this buffer. The argument is an array of three 16-bit integers. The above C-code example sets the value of X_0 to $1234ABCD330E_{16}$.

The pointer to the previous value of x_i is useful if a restart from that point is desired at a later time. The 48-bit X_i value must be copied out

of the internal buffer before seed48 is called again or it will be destroyed. The following code sequence, for example, restarts a program with a saved value of X_i.

```
short int newx[3], oldx[3], *shp, *seed48( ), i;
shp = seed48(newx); /* initialize with whatever is in newx */
for (i = 0; i < 3; i++) oldx[i] = shp[i]; /* save previous Xi in
oldx */
    ....
    ....
    ....
seed48(oldx); /* reinitialize with oldx */
```

3.7 Function lcong48

Example C usage:

```
short int param[7];
param[0] = 0x330E; param[1] = 0xABCD; parem [2] = 0x1234;
param[3] = 0xE66D; param[4] = 0xDEEC; param[5] = 0x5;
param[6] = 0xB;
lcong48(param);
```

This is an initialization entry point that sets the values of X_0, a, and c; hence, different 48-bit linear congruential generators may be created by specifying different values for the multiplier a and the addend c. The argument passed to lcong48 is an array of seven 16-bit integers. The first three specify a 48-bit value of X_0; the next three specify a 48-bit value of the multiplier a, and the last one specifies a 16-bit value of the addend c. Hence, the above C-code examples set $X_0 = 1234ABCD330E_{16}$, $a = 5DEECE66S_{16}$, and $c = B_{16}$.

IV. TIMING RESULTS

Table III presents the time required to generate, using function drand48, 10^6 pseudo-random numbers on five different computer hardware configurations. More precisely, Table III gives the time required to execute the following short C-language program:

```
main( ) {
register int i, j, h;
```

Table III—Time required to generate 10^6 pseudo-random numbers

Computer	Time (sec)
PDP-11/45	440
PDP-11/45 with Fabritek cache	340
PDP-11/70	162
VAX-11/750	200
VAX-11/780	96

```
double nfd, drand48( );
int lli, llj;
short int nn[500];
long int seedval;
seedval = 0x1234ABCD; srand48(seedval);
for(i = 0; i < 500; i++) nn[i] = 0;
lli = 1000; llj = 1000;
nfd = 500;
for(i = 0; i < lli; i++) for(j = 0; j < llj; j++)
  {h = nfd*drand48( );
    nn(h)++;
  }
}
```

Execution of drand48 accounts for 80 percent, approximately, of the times listed in Table III. For the timing tests on the PDP-11/45 and PDP-11/70, our version of drand48 that employs the floating-point arithmetic instructions was used, since it is substantially faster than the version that employs the integer arithmetic instructions. The first three entries in Table III represent separate executions of the same binary machine code; hence, the time differences reflect brute-force speed differences of the processor and/or memory hardware. The same is true for the last two entries in Table III. The comparison between the PDP-11 and the VAX-11 is, however, more subtle since a complete recoding of drand48 is involved here. Essentially, when comparing the VAX-11 with the PDP-11 in Table III, we are comparing the time required to accomplish the identical "function" on both computers using a near optimally coded version of drand48 on each machine. The faster time for the VAX-11/780 must be attributed to at least two factors: a more powerful and capable instruction set than the PDP-11, and faster basic hardware.

V. DISCUSSION

We have described the design of drand48, a good, pseudo-random number-generator subroutine, and presented enough output data so that this subroutine can be recoded for a new processor and quickly tested for bugs. While the reproduction by a new drand48 implementation of either Table I or II surely is not a logically complete test of correctness, it is, at least, a first-order indication that will catch many of the common types of programming errors. The drand48 subroutine is not a panacea, an "ultimate" generator for all purposes. There are numerous other algorithms for generating pseudo-random sequences, and each one has its advantages, disadvantages, advocates, and detractors. The most sophisticated requirements will always have to be met by custom design of the generator and extensive statistical testing.

However, it is not at these most sophisticated users that drand48 is aimed.

We live in an age when new computers are coming into existence very rapidly. The *UNIX* operating system and the C programming language are already running on the Western Electric 3B20 processor, and the 3B05 is not far behind. Other new processors have been or are being introduced by other manufacturers. As existing programs migrate to these new machines, it would be desirable for the associated pseudo-random number generator to continue to produce the same output sequences. A subroutine written in a portable, high-level programming language is one possible solution, and drand48 has proven itself amenable to such an approach. However, for applications in which speed is of paramount importance, subroutines written in assembly language are useful. (The implementation of drand48 in the "portable" dialect of C ran significantly slower[7] than the assembly language versions described in this paper.) The drand48 subroutine has worked well on the PDP-11 and VAX-11 computers; the author hopes that the information provided in this paper will make it possible for new implementations of drand48 and its variants to be accurately coded for at least some of the new computers that will ultimately become popular in the future.

VI. ACKNOWLEDGMENT

The author wishes to thank Lawrence Rosler for providing him with information on the portable C version of drand48 and for his helpful comments on preliminary drafts of this paper.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Reading, MA: Addison-Wesley, 1969.
2. R. R. Coveyou and R. D. MacPherson, "Fourier Analysis of Uniform Random Number Generators," J. Assoc. Comp. Mach., *14*, No. 1 (January 1967), pp. 100–19.
3. *PDP-11/70 Processor Handbook*, Maynard, MA: Digital Equipment Corporation, 1976.
4. *VAX-11/780 Architecture Handbook*, Vol. 1, Maynard, MA: Digital Equipment Corporation, 1977.
5. L. Rosler and N-P. Nelson, unpublished work.
6. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
7. L. Rosler, private communication.